High-Performance General Solver for Extremely Large-Scale Semidefinite Programming Problems

Katsuki Fujisawa
Chuo University & JST CREST
1-13-27, Kasuga, Bunkyo-ku, Tokyo, Japan
Email: fujisawa@indsys.chuo-u.ac.jp

Toshio Endo
Tokyo Institute of Technology & JST CREST
2-12-1, Ookayama, Meguro-ku, Tokyo, Japan
Email: endo@is.titech.ac.jp

Hitoshi Sato
Tokyo Institute of Technology & JST CREST
2-12-1, Ookayama, Meguro-ku, Tokyo, Japan
Email: hitoshi.sato@gsic.titech.ac.jp

Makoto Yamashita
Tokyo Institute of Technology
2-12-1, Ookayama, Meguro-ku, Tokyo, Japan
Email: Makoto.Yamashita@is.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology, JST CREST & NII
2-12-1, Ookayama, Meguro-ku, Tokyo, Japan
Email: matsu@is.titech.ac.jp

Maho Nakata
RIKEN
2-1, Hirosawa, Wako, Saitama, Japan
Email: maho@riken.jp

***Abstract*—Semidefinite programming (SDP) is one of the most important problems among optimization problems at present. It is relevant to a wide range of fields such as combinatorial optimization, structural optimization, control theory, economics, quantum chemistry, sensor network location and data mining. The capability to solve extremely large-scale SDP problems will have a significant effect on the current and future applications of SDP. In 1995, Fujisawa et al. started the SDPA(Semidefinite programming algorithm) Project aimed at solving large-scale SDP problems with high numerical stability and accuracy. SDPA is one of the main codes to solve general SDPs. SDPARA is a parallel version of SDPA on multiple processors with distributed memory, and it replaces two major bottleneck parts (the generation of the Schur complement matrix and its Cholesky factorization) of SDPA by their parallel implementation. In particular, it has been successfully applied to combinatorial optimization and truss topology optimization. The new version of SDPARA (7.5.0-G) on a large-scale supercomputer called TSUBAME 2.0 at the Tokyo Institute of Technology has successfully been used to solve the largest SDP problem (which has over 1.48 million constraints), and created a new world record. Our implementation has also achieved 533 TFlops in double precision for large-scale Cholesky factorization using 2,720 CPUs and 4,080 GPUs.**

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming, GPU Computing; J.0 [**General**]: Combinatorial Optimization, Structural Optimization, Quantum Chemistry; G.1.6 [**Optimization**]: Convex Programming, Nonlinear Programming

General Terms

Parallel Computing, Optimization, Applications

Keywords

GPGPU, Dense Matrix Algebra, Semidefinite Programming, General Optimization Solver

I. Introduction

Semidefinite Programming (SDP) is a subfield of mathematical programming and is used to optimize an objective function over positive semidefinite matrices. SDP has been regarded as one of the most important optimization problems for several reasons.

1) SDP is used in a wide range of fields such as combinatorial optimization, structural optimization, control theory, economics, quantum chemistry, sensor network location, data mining and machine learning [1], [2].
2) SDP theoretically includes a large number of convex programming such as linear programming (LP), convex quadratic programming (QP), and second-order cone programming (SOCP). For example, LP is a special case of SDP when all the matrices involved are diagonal. Therefore, we can convert typical convex programming problems to SDP problems.
3) SDP relaxation techniques can be used to obtain strong upper and lower bounds. This is true not only for convex optimization problems but also for nonconvex optimization problems, which are encountered in application areas such as control theory and smart grids and systems.
4) There exist several fast and stable theoretical algorithms [1]. For example, we can obtain an optimal solution in polynomial time using a primal-dual interior-point method (PDIPM).
5) Many state-of-the-art multithread-based parallel software packages for SDP, such as SDPA [3], [4], [5], CSDP [6], SeDuMi [7], and SDPT3 [8] have been developed. In addition, there exist MPI-based parallel software packages for supercomputers such as SDPARA [9],

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

[10], PCSDP [13], and PDSDP [14].

Solving extremely large-scale SDP problems is considered to be important and challenging. For example, it would be useful to obtain the optimal solution of the quadratic assignment problem (QAP), one of the most difficult NP-hard problems. If we could solve an SDP problem with more than 10^6 constraints, we could obtain a strong lower bound of the QAP with a problem size greater than or equal to 40.

For solving SDP problems in polynomial time, the primal-dual interior-point method (PDIPM) [15] is well known as a powerful and stable method. In 1995, Fujisawa et al. started the SDPA Project [5] aimed at solving large-scale SDP problems with high numerical stability and accuracy. Semidefinite programming algorithm (SDPA) [5] is one of the main libraries for solving general SDP problems that have no special structures such as a rank-1 condition, and an optimization software package is used along with the PDIPM for solving the standard SDP problem (formulation (1) in Section II). SDPA incorporates special data structures for handling block diagonal data matrices, and efficient techniques for computing search directions when problems become large scale and/or sparse [4]. SDPA achieves good performance on a single processor [5]; however, when the size of an SDP problem increases, the use of the PDIPM introduces two major performance bottlenecks in each iteration, even if we exploit the sparsity of the data matrices: the computation of the Schur complement matrix (SCM) and the Cholesky factorization of the SCM.

The semidefinite programming algorithm parallel version (SDPARA) [9], [10] is a parallel implementation of the SDPA library for multiple processor systems with distributed memory. In this implementation, performance bottlenecks are relaxed by parallelization techniques using MPI, OpenMP, ScaLAPACK [11], BLACS, optimized BLAS, and MUMPS [12]. Each process reads the input data and stores them and all variables in the process memory space, while the SCM data are divided between processes. Thus, SDPARA can compute each row of the SCM in parallel, and apply the parallel Cholesky factorization provided by ScaLAPACK to the SCM. We have reported that SDPARA achieves high scalability for large-scale SDP problems [3], [9], [10] on supercomputers and have confirmed that SDPARA is much faster than other parallel implementations such as PCSDP [13] and PDSDP [14] when solving large-scale sparse SDP problems [9], [10]. By using existing SDPARA (version 7.4.0), we have successfully solved large-scale SDP problems with a sparse SCM and a size of about $300,000$, and created a world record in 2011 [3].

The target of this study is highly challenging: in order to accommodate problems related to the QAP and truss topology optimization, our objective is to solve SDP problems with a dense SCM and a size larger than 10^6. Aside from an increase in problem sizes, changes in the characteristics of problems (the density of the SCM) also pose new challenges. Let m be the SCM size (which equals the number of constraints) and n be the size of the data matrix. For the first bottleneck part (computation of the SCM), we have previously developed an algorithm that decreases its time complexity $O(mn^3 + m^2n^2)$ to $O(m^2)$ by exploiting the sparsity of the data matrix [9]. However, if the SCM is large and dense, the second bottleneck part (Cholesky factorization of the SCM), whose complexity is $O(m^3)$, dominates the entire execution time. In this study, we accelerate this part by using massively parallel GPUs with computational performance much higher than that of CPUs. In order to achieve scalable performance with thousands of GPUs, we utilize a high-performance BLAS kernel along with optimization techniques to overlap computation, PCI-Express communication, and MPI communication.

After the improvement of the performance of the two bottleneck parts, we have noticed that other components that are harmless when solving smaller problems degrade the performance on large-scale supercomputers; one such component is the initialization phase of the solver. Further, the data reconstruction for the SCM doubles the memory usage of SDPARA, which adversely affect solving of large-scale problems.

This paper describes the design and implementation of a new version of SDPARA (version 7.5.0-G), which includes the following features:

1) We halve the computational memory space for the computation of the first bottleneck part (Section V).
2) We parallelize the preprocessing of the problem and the estimation of the cost function (Sections V and VIII).
3) For large and dense problems, we accelerate the Cholesky factorization by using thousands of GPUs efficiently. We also introduce techniques to overlap the communication and computation (Section VII).
4) We employ the ILP64 programming model for extremely large-scale matrix computation. We partially modify source codes of SDPARA, MUMPS, and ScaLAPACK.

We have conducted the performance evaluation of SDPARA 7.5.0-G on TSUBAME 2.0, which is a peta-scale GPU-accelerated supercomputer at the Tokyo Institute of Technology. In the evaluation, we solved the largest SDP problem (which has over 1.48 million constraints), and created a new world record in 2012. Our implementation also achieved 533 TFlops in double precision for large-scale Cholesky factorization using 4,080 GPUs.

The source codes and user manuals of SDPA and SDPARA are available at the SDPA Project Web site:

http://sdpa.sourceforge.net/

II. Semidefinite Programming

The standard form SDP is the following primal-dual form.

$$\begin{array}{lcll} \mathcal{P} & : & \text{minimize} & \sum_{k=1}^{m} c_k x_k \\ & & \text{subject to} & \boldsymbol{X} = \sum_{k=1}^{m} \boldsymbol{F}_k x_k - \boldsymbol{F}_0, \quad \boldsymbol{X} \succeq \boldsymbol{O}. \\ \mathcal{D} & : & \text{maximize} & \boldsymbol{F}_0 \bullet \boldsymbol{Y} \\ & & \text{subject to} & \boldsymbol{F}_k \bullet \boldsymbol{Y} = c_k \ (k = 1, \ldots, m), \quad \boldsymbol{Y} \succeq \boldsymbol{O}. \end{array} \tag{1}$$

We use $\mathbb{S}^n$ for the space of $n \times n$ symmetric matrices. The notation $\boldsymbol{X} \succeq \boldsymbol{O}$ ($\boldsymbol{X} \succ \boldsymbol{O}$) indicates that $\boldsymbol{X} \in \mathbb{S}^n$ is

a positive semidefinite (positive definite) matrix. The inner-product between $\boldsymbol{U} \in \mathbb{S}^n$ and $\boldsymbol{V} \in \mathbb{S}^n$ is defined by $\boldsymbol{U} \bullet \boldsymbol{V} = \sum_{i=1}^{n} \sum_{j=1}^{n} U_{ij} V_{ij}$.

In most SDP applications, it is common for the input data matrices $\boldsymbol{F}_0, \ldots, \boldsymbol{F}_m$ to share the same diagonal block structure $(n_1, \ldots, n_h)$. Each input data matrix $\boldsymbol{F}_k$ $(k = 1, \ldots, m)$ consists of sub-matrices in the diagonal positions, as follows:

$$\boldsymbol{F}_k = \begin{pmatrix} \boldsymbol{F}_k^1 & \boldsymbol{O} & \boldsymbol{O} & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{F}_k^2 & \boldsymbol{O} & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{O} & \ddots & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{O} & \boldsymbol{O} & \boldsymbol{F}_k^h \end{pmatrix}$$

where $\boldsymbol{F}_k^1 \in \mathbb{S}^{n_1}, \boldsymbol{F}_k^2 \in \mathbb{S}^{n_2}, \ldots, \boldsymbol{F}_k^h \in \mathbb{S}^{n_h}$.

Note that $\sum_{\ell=1}^{h} n_\ell = n$ and the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ share the same block structure. We define $n_{\max}$ as $\max\{n_1, \ldots, n_h\}$. For the blocks where $n_\ell = 1$, the constraints of positive semidefiniteness are equivalent to the constraints of the nonnegative orthant. Such blocks are sometimes called LP blocks.

The size of given SDP problems can be roughly measured in terms of five numbers:

1) m: the number of equality constraints in the dual form $\mathcal{D}$ (which equals the size of the SCM)
2) n: the size of the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$
3) $n_{\max}$: the size of the largest block of input data matrices
4) nnz: the total number of nonzero elements in all data matrices

Figure 1 shows the largest SDP problem, which is solved in Section VII, and its block diagonal structure. The parameters $m, n, n_{\max}$, and nnz in this SDP problem are 1,484,406, 1,779,204, 1,682, and 23,476,376, respectively.

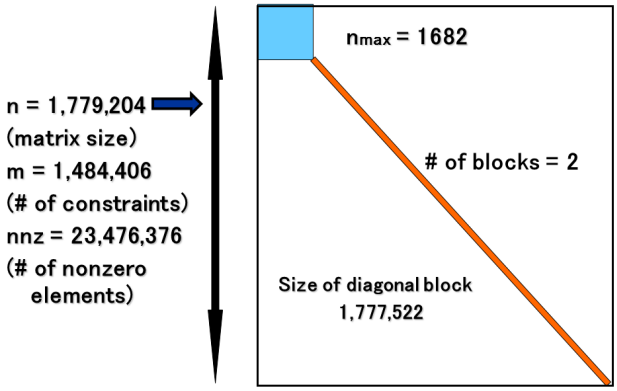


Fig. 1. Largest SDP problem and its block diagonal structure

III. Important Applications of SDP

As mentioned in Section I, we can easily identify important applications of SDP in many research areas. Sets of SDP benchmark problems are helpful for the development and evaluation of SDP codes. There exist some standard benchmark test sets of SDP instances: SDPLIB [1], the DIMACS test sets [2], and benchmark sets maintained by Hans Mittelmann [3] and Renata Sotirov [4]. In this paper, we have selected three types of large-scale SDP instances associated with recent applications of SDP in truss topology design, combinatorial optimization, and quantum chemistry; however, SDPARA can also achieve high performance when solving large-scale SDP problems in other important application areas.

A. Truss Topology Design Problems

Consider a truss with fixed locations of nodes and members. The truss topology design problem formulated as an SDP problem [16] is to find a truss of minimum volume such that the minimum fundamental frequency of vibration is greater than or equal to the predetermined critical value. The vector of member cross-sectional areas and the vector of bar lengths are denoted by $\mathbf{A}(=\{A_i\})$ and $\mathbf{L}(=\{L_i\})$ respectively.

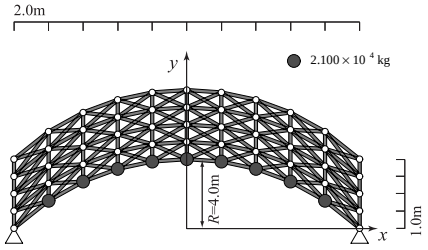


Fig. 2. Plane arch grid

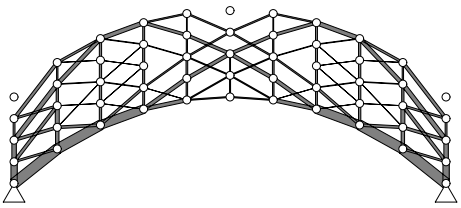


Fig. 3. Optimal topology of the plane arch grid

The primal and dual SDP problems for this case are formulated as

$$\begin{aligned} \mathcal{P}': \quad & \text{minimize} && \sum_{i=1}^{m} A_i L_i \\ & \text{subject to} && \mathbf{X} = \sum_{i=1}^{m} (\mathbf{K}_i - \bar{\Omega}\mathbf{M}_i) A_i - \bar{\Omega}\mathbf{M}_0, \quad (2) \\ & && \mathbf{X} \in \mathcal{S}^n, \mathbf{X} \succeq \mathbf{O}, \\ & && A_i \geq 0 \ (i = 1, 2, \cdots, N^m). \\ \mathcal{D}': \quad & \text{maximize} && \bar{\Omega}\mathbf{M}_0 \bullet \mathbf{Y} \\ & \text{subject to} && (\mathbf{K}_i - \bar{\Omega}\mathbf{M}_i) \bullet \mathbf{Y} = L_i, \quad (3) \\ & && \mathbf{Y} \in \mathcal{S}^n, \mathbf{Y} \succeq \mathbf{O}, \end{aligned}$$

where $\bar{\Omega}$ is a lower bound on the fundamental frequency of vibration of the truss and $\mathbf{M}_0$ is the nonstructural mass

[1] http://euler.nmt.edu/~brian/sdplib/sdplib.html
[2] http://dimacs.rutgers.edu/Challenges/Seventh/Instances/
[3] http://plato.asu.edu/ftp/sparse_sdp.html
[4] http://lyrawww.uvt.nl/~sotirovr/libarary/

matrix. Let $\mathbf{K}$ and $\mathbf{M}_s$ denote the stiffness matrix and the structural mass matrix, both of which are functions of $\mathbf{A}$. The matrices $\mathbf{K}_i$ and $\mathbf{M}_i$ are the stiffness and mass matrices of bar i, respectively. If we consider the plane arch grid shown in Figure 2, we can obtain an optimal topology (Figure 3).

B. Quadratic Assignment Problems

The QAP is a fundamental and important combinatorial optimization problem that involves finding a permutation for given flow and distance matrices, and it is formulated as follows:

$$\begin{array}{ll} \text{minimize} & \text{trace}(AXBX^T) + \text{trace}(CX) \\ \text{subject to} & XX^T = X^TX = I_n, \end{array} \tag{4}$$

where A and B are $n\times n$ real symmetric matrices, C is an $n\times n$ real matrix, and I_n is an $n \times n$ identity matrix. For finding a lower bound or the exact value of the QAP in (4), we consider the following doubly nonnegative (DNN) relaxation problem [17]:

$$\begin{array}{ll} \text{minimize} & \left\langle \begin{pmatrix} 0 & \text{vec}(C)^T \\ \text{vec}(C) & B^T \otimes A \end{pmatrix}, \begin{pmatrix} y_{00} & y^T \\ y & Y \end{pmatrix} \right\rangle \\ \text{subject to} & Y = (Y^{ij})_{1\le i,j\le n} \in \mathbb{S}_+^{n^2}, \\ & Y^{ij} = (Y^{ij}_{k\ell})_{1\le k,\ell\le n} \in \mathbb{S}^n \ (i,j=1,\ldots,n), \\ & y = (y^i)_{1\le i\le n}, y^i_k = Y^{ii}_{kk} \ (i,k=1,\ldots,n), \\ & y_{00} = 1, \\ & Y^{ij}_{k\ell} \ge 0 \ (i,j,k,\ell=1,\ldots,n), \\ & \sum_{i=1}^n Y^{ii} = I_n, \\ & \langle I_n, Y^{ij}\rangle = \delta_{ij} \ (1\le i\le j\le n), \\ & \langle J_n, Y^{ij}\rangle = 1 \ (1\le i\le j\le n), \end{array} \tag{5}$$

where δ_{ij} is a Kronecker delta, J_n is an $n \times n$ matrix of all ones, and $\text{vec}(C)$ is the vector formed from the columns of matrix C. We can obtain the optimal value and solution of (5) by applying SDP solvers since (5) can be reformulated into an SDP problem. Before solving (5) with the SDP solvers, we apply a facial reduction algorithm based on the study of Zhao et al. [18] to (5) since (5) is too degenerate to get an accurate value and solution. Indeed, (5) does not have any interior feasible solutions, and thus, it is difficult to obtain an accurate value and solution by using SDP solvers in which primal-dual interior-point methods are implemented. In addition, we rewrite the resulting DNN problem as a linear matrix inequality described by variables $Y^{ij}_{k\ell}$ to improve the numerical stability of the computation of SDP solvers. For instance, we can write $\langle I_n, Y^{ij}\rangle = \delta_{ij}$ and $\langle J_n, Y^{ij}\rangle = 1$, respectively, as follows:

$$\sum_{k=1}^n Y^{ij}_{kk} = \delta_{ij} \text{ and } \sum_{k,\ell=1}^n Y^{ij}_{k\ell} = 1.$$

C. SDP Relaxation of Electronic Structure Problems for Atoms and Molecules

The ultimate goal in chemistry is to know the exact wave-function by solving the Schrödinger equation

$$H\Psi = E\Psi, \tag{6}$$

where H is the Hamiltonian of the system, E is the energy, and Ψ is the wave function [19]. We can know the Hamiltonian of the systems quite easily, so if we solve the Schrödinger equation, then we can obtain a lot of information: how superconductivity occurs, how a protein works as an enzyme, how CO_2 is converted to O_2, etc.

However, it is also known that solving the Schrödinger equation is difficult because it involves solving the eigenvalue problem in the form of a partial differential equation of second order. The task is difficult even if we employ localized atomic orbitals to discretize the problem, as it becomes an eigenvalue problem involving a huge matrix. Therefore, we are looking at an alternative method; we are interested in the direct determination of the second-order reduced density matrix

$$\Gamma^{i_1i_2}_{j_1j_2} = \frac{1}{2}\langle\Psi|a^\dagger_{i_1}a^\dagger_{i_2}a_{j_1}a^\dagger_{j_2}|\Psi\rangle.$$

If we restrict ourselves to the ground-state problem, in which the lowest eigenvalue and its eigenvector are found, the problem is further simplified to the minimization of the linear functional:

$$E_{\text{g}} = \underset{\Gamma\in\mathcal{P}}{\text{Min}}\, \text{Tr}H\Gamma. \tag{7}$$

This simplification is quite drastic as the number of variables reduces from $O(r!)$ to $O(r^4)$, where r is the number of localized atomic orbitals. The barter for such a drastic simplification is that we have to know about the N-representability condition [20]; otherwise, the calculated energy becomes much lower than the exact one. Therefore, for actual calculations, we just minimize under sufficient conditions like P, Q, and G conditions [20], [21].

Eq. (6) can be cast as an SDP problem for the ground state.

$$\begin{cases} \inf & \langle H, \Gamma^{\text{full}}\rangle \\ \text{s.t.} & \langle I, \Gamma^{\text{full}}\rangle = N, \\ & 0 \preceq \Gamma^{\text{full}}, \end{cases}$$

where N is the number of electrons in the system and Γ^{full} is the von Neumann or Landau density matrix. Eq. (7) can also be cast as an SDP problem [22], [23].

$$\begin{cases} \inf & \langle H, \Gamma\rangle \\ \text{s.t.} & 0 \preceq \Gamma, \text{etc. (“}N\text{-representability conditions”).} \end{cases}$$

Nakata et al. [22] formulated the problem as a primal SDP problem. They performed a direct variational calculation of the 2-reduced density matrix (2-RDM) by employing the P, Q, and G conditions and by using the well-established semidefinite programming solver known as SDPA [4], [5]. This approach was applied to many few-electron atoms and molecules. Nakata et al.'s [22] results obtained by using the P, Q, and G conditions were very encouraging; they obtained around 100 to 130% of correlation energy and could produce a dissociation curve of the nitrogen dimer that was in good agreement with full CI results. Zhao et al. included the $T1$ and $T2$ conditions in addition to the P, Q, and G conditions in the above-mentioned approach in their calculations on small molecules [23].

IV. Basic framework of the primal-dual interior-point method

In this section, we explain the basic framework of the PDIPM on which SDPA and SDPARA are based. The most important theoretical aspects of the PDIPM is that it solves both primal and dual forms simultaneously and finds their optimal solution in polynomial time.

The Karush-Kuhn-Tucker (KKT) conditions theoretically guarantee that a point $(\boldsymbol{x}^*, \boldsymbol{X}^*, \boldsymbol{Y}^*)$ satisfying the system in (8) below is an optimal solution of (1) when the so-called Slater's condition is satisfied.

$$\text{KKT} \begin{cases} \boldsymbol{X}^* = \sum_{k=1}^m \boldsymbol{F}_k x_k^* - \boldsymbol{F}_0, \\ \boldsymbol{F}_k \bullet \boldsymbol{Y}^* = c_k \ (k = 1, 2, \ldots, m), \\ \sum_{k=1}^m c_k x_k^* = \boldsymbol{F}_0 \bullet \boldsymbol{Y}^*, \\ \boldsymbol{X}^* \succeq \boldsymbol{O}, \boldsymbol{Y}^* \succeq \boldsymbol{O}. \end{cases} \tag{8}$$

As we have shown in Figure 4, in the PDIPM, the algorithm starts from a feasible or infeasible point. In each iteration, it computes the search direction $(d\boldsymbol{x}, d\boldsymbol{X}, d\boldsymbol{Y})$ from the current point towards the optimal solution, decides the step size, and advances by the step size in the search direction. If the current point reaches a small neighborhood of the optimal solution, the PDIPM terminates the iteration and returns the approximate optimal solution. The PDIPM is described in many papers (See [24], [25], [15], [26], and [27]). The framework we use relies on the HRVW/KSH/M approach [25], [15], [26], and we will use the appropriate norms $||\cdot||$ for matrices and vectors.

Algorithmic Framework of PDIPM

Step 0 : Choose a feasible or infeasible initial point $(\boldsymbol{x}^0, \boldsymbol{X}^0, \boldsymbol{Y}^0)$ such that $\boldsymbol{X}^0 \succ \boldsymbol{O}$ and $\boldsymbol{Y}^0 \succ \boldsymbol{O}$. Set the centering parameter $\beta \in (0, 1)$, the boundary parameter $\gamma \in (0, 1)$, the threshold parameter $\epsilon > 0$, and the iteration number $s = 0$.

Step 1 : Compute the residuals of the primal feasibility $\boldsymbol{P}$, the dual feasibility $\boldsymbol{d}$, and the primal-dual gap g:

$$\begin{cases} \boldsymbol{P} &= \boldsymbol{F}_0 - \sum_{k=1}^m \boldsymbol{F}_k x_k^s + \boldsymbol{X}^s, \\ d_k &= c_k - \boldsymbol{F}_k \bullet \boldsymbol{Y}^s \quad (k = 1, \ldots, m), \\ g &= \sum_{k=1}^m c_k x_k^s - \boldsymbol{F}_0 \bullet \boldsymbol{Y}^s. \end{cases}$$

If $\max\{||\boldsymbol{P}||, ||\boldsymbol{d}||, |g|\} < \epsilon$ (namely, all residuals above are sufficiently small), stop the iteration and output $(\boldsymbol{x}^s, \boldsymbol{X}^s, \boldsymbol{Y}^s)$ as an approximate optimal solution.

Step 2 : Compute the the search direction $(d\boldsymbol{x}, d\boldsymbol{X}, d\boldsymbol{Y})$.

Step 2a: Compute the SCM $\boldsymbol{B}$ by the formula

$$B_{ij} = \left((\boldsymbol{X}^s)^{-1} \boldsymbol{F}_i \boldsymbol{Y}^s \right) \bullet \boldsymbol{F}_j. \tag{9}$$

Step 2b: Apply the Cholesky factorization to $\boldsymbol{B}$ and obtain a lower triangular matrix $\boldsymbol{L}$ such that $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^T$.

Step 2c: Obtain a component of the search direction $d\boldsymbol{x}$ by solving the equations

$$\boldsymbol{L}\widetilde{d\boldsymbol{x}} = \boldsymbol{r} \text{ and } \boldsymbol{L}^T d\boldsymbol{x} = \widetilde{d\boldsymbol{x}}$$

for the right-hand-side vector $\boldsymbol{r}$ computed as

$$r_k = -d_k + \boldsymbol{F}_k \bullet \left((\boldsymbol{X}^s)^{-1} (\boldsymbol{R} + \boldsymbol{P}\boldsymbol{Y}^s) \right) \quad (k = 1, \ldots, m), \tag{10}$$

where $\boldsymbol{R} = \beta\mu\boldsymbol{I} - \boldsymbol{X}^s\boldsymbol{Y}^s$ with $\mu = \boldsymbol{X}^s \bullet \boldsymbol{Y}^s / n$.

Step 2d: Compute the remaining components of the search direction $(d\boldsymbol{X}, d\boldsymbol{Y})$ as follows:

$$\begin{cases} d\boldsymbol{X} = -\boldsymbol{P} + \sum_{k=1}^m \boldsymbol{F}_k dx_k, \\ \widetilde{d\boldsymbol{Y}} = (\boldsymbol{X}^s)^{-1}(\boldsymbol{R} - d\boldsymbol{X}\boldsymbol{Y}^s), \\ d\boldsymbol{Y} = (\widetilde{d\boldsymbol{Y}} + \widetilde{d\boldsymbol{Y}}^T)/2. \end{cases}$$

Step 3 : Maximize the step sizes (lengths) such that the following positive definiteness conditions are satisfied.

$$\begin{cases} \alpha_P &= \max\{\alpha \in [0, 1] : \boldsymbol{X}^s + \alpha d\boldsymbol{X} \succeq \boldsymbol{O}\}, \\ \alpha_D &= \max\{\alpha \in [0, 1] : \boldsymbol{Y}^s + \alpha d\boldsymbol{Y} \succeq \boldsymbol{O}\}. \end{cases}$$

Step 4 : Update the current point as follows:

$$(\boldsymbol{x}^{s+1}, \boldsymbol{X}^{s+1}, \boldsymbol{Y}^{s+1}) \leftarrow (\boldsymbol{x}^s, \boldsymbol{X}^s, \boldsymbol{Y}^s) + \gamma(\alpha_P d\boldsymbol{x}, \alpha_P d\boldsymbol{X}, \alpha_D d\boldsymbol{Y}).$$

Set $s \leftarrow s + 1$, and return to Step 1.

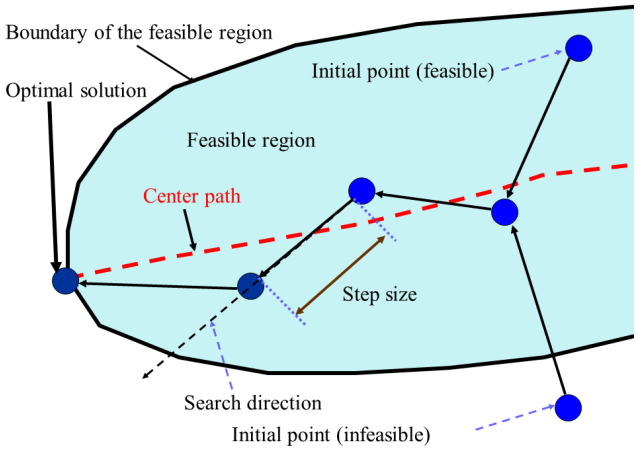


Fig. 4. Primal-dual interior-point method

Steps 2a and 2b correspond to the first and second bottleneck parts defined in Section I, respectively. We shall define the computation in Steps 2a and 2b as ELEMENTS and CHOLESKY, respectively. As indicated in [9], ELEMENTS ($O(mn^3 + m^2n^2)$) and CHOLESKY ($O(m^3)$) have often accounted for 80% to 90% of the total execution time of the PDIPM. Therefore, researchers have focused on reducing the time taken for these steps [5], [6], [7], [8]. We have reduced the time complexity of ELEMENTS from $O(mn^3 + m^2n^2)$ to $O(m^2)$ when solving the sparse SDP problem [4].

Table I shows that the execution time of existing SDPARA (version 7.4.0) for two large-scale SDP problems. ELEMENTS accounted for about 97.5% of the total execution time (TOTAL) when solving the Electronic1 problem, while

TABLE I
EXECUTION TIME (IN SECONDS) OF SDPARA 7.4.0 FOR LARGE-SCALE SDP PROBLEMS

Name	n	m	ELEMENTS	CHOLESKY	TOTAL
Electronic1	33,178	76,554	650,413	6,912	666,578
QAP1	486,600	379,350	3,242	343,406	350,372

TABLE II
PERFORMANCE COMPARISON AMONG SDPARA 7.4.0, PCSDP 1.0, AND PDSDP 5.8

Name	SDPARA (s)	PCSDP (s)	PDSDP (s)
Electronic2	173	1,067	948
Electronic3	221	527	540
QAP2	29	139	662

CHOLESKY accounted for about 98% of TOTAL when solving the QAP1 problem. We carry out a performance comparison (Table II) with other MPI-based parallel software packages (PCSDP [13] and PDSDP [14]) to show the performance superiority of SDPARA (version 7.4.0). We have performed all numerical experiments, whose results are given in Tables I, II, III, and IV on the PC cluster of Chuo University (hereafter referred to as "the OPT cluster"). The OPT cluster consists of 16 nodes, 32 CPUs (Intel Xeon X5670 2.93 GHz), and 192 cores. Each node has 128 GB of DDR3 memory and a QDR 40 Gbps InfiniBand HCA.

V. NEW IMPLEMENTATION OF SDPARA AND REDUCTION OF THE COMPUTATIONAL MEMORY

This section explains the new implementation of SDPARA (version 7.5.0) for improving the performance and reducing the computational memory space.

We have already mentioned that SDPARA has two major bottleneck parts (ELEMENTS and CHOLESKY). However, there are some candidates for new bottleneck parts when solving extremely large-scale SDP problems and using a large number of processors. Let N be the number of available processors, and assign numbers (1 to N) to all processors. SDPARA [9], [10] first reads the input data $m, n, \boldsymbol{c}, \boldsymbol{F}_0, \boldsymbol{F}_1, \ldots, \boldsymbol{F}_m$ into processor 1 and distributes all the data among processors 1 to N through the network device. The execution time of this initialization is often over 10% of the total execution time of SDPARA when more than several hundreds processors are used, and therefore, we store the input data in a local storage device of each node in advance so that each node can read the input data without having to communicate with other nodes. As explained in Section VI, each node of the TSUBAME 2.0 supercomputer has one local SSD volume, and we accelerate the initialization by over ten times by utilizing the local storage devices.

SDPARA has another initialization phase(hereafter "INIT"), which may be a new bottleneck part. As discussed in [4], some cost functions used for investigating the sparsity of the problem must be evaluated once after reading the input data. While each processor evaluates all cost functions in SDPARA 7.4.0, in SDPARA 7.5.0, the execution time of INIT is significantly reduced by applying parallel computing techniques (see Section VIII for more details).

Here we describe how to reduce the computational memory space for the computation of ELEMENTS and CHOLESKY. The SCM B is stored on distributed memory in two different ways [9], [10]; all elements of each row of $\boldsymbol{B}$ are computed and stored temporarily on a single processor (hence each processor computes multiple rows of $\boldsymbol{B}$ independent of the other processors) (Figure 5), and they are then redistributed according to the two-dimensional block-cyclic distribution by calling the function `Cpdgemr2d` of ScaLAPACK for parallel Cholesky factorization (Figure 6). The advantage of this technique has already been discussed in [3], [9], and [10]. However, the computational memory space required is double that in the case in which we have only the two-dimensional block-cyclic distribution of the SCM. If we solve the large-scale SDP problem (Figure 1) on TSUBAME 2.0 using SDPARA 7.4.0, we need to allocate about MEM_SIZE memory to each node for storing two types of SCMs, where MEM_SIZE indicates 32,834 Mbytes / # of nodes we can use. This implies that over about 800 nodes will be required for solving the SDP problems (Figure 1) since each node of TSUBAME 2.0 has 54 Gbytes memory and other data $\boldsymbol{c}, \boldsymbol{F}_0, \boldsymbol{F}_1, \ldots, \boldsymbol{F}_m, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{x}$ must be also stored. SDPARA 7.5.0 requires only one matrix $(m \times m)$ for computing the SCM, and it directly computes elements of the two-dimensional block-cyclic distribution of the SCM. Therefore, about 400 nodes are required for solving the SDP problem (Figure 1).

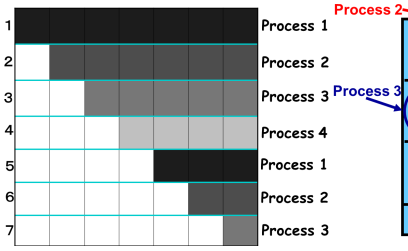


Fig. 5. Row-wise Distribution of the Schur complement matrix $B(7 \times 7)$ using 4 processes in SDPARA 7.4.0

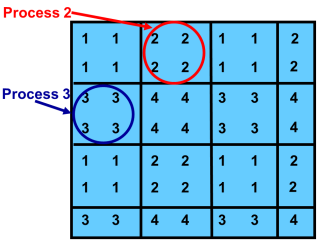


Fig. 6. Two-dimensional block-cyclic distribution of the Schur complement matrix $B(7 \times 7)$ using 4 processes

TABLE III
TOTAL REQUIRED MEMORY SPACE FOR EACH NODE

Name	n	n_{max}	m	7.4.0 (bytes)	7.5.0 (bytes)
Electronic4	4,444	1,064	4,743	365 M	354 M
QAP3	104,300	362	72,600	5.3 G	2.6 G

TABLE IV
EXECUTION TIME (IN SECONDS) OF ELEMENTS AND CHOLESKY

Name	ELEMENTS (7.4.0)	ELEMENTS (7.5.0)	CHOLESKY
Electronic4	124	167	5
QAP3	74	110	4,210

Tables III and IV show the total memory space required

for each node and the execution time of ELEMENTS and CHOLESKY on the OPT cluster, respectively. The execution time of ELEMENTS slightly increases when using SDPARA 7.5.0 and solving the QAP3 problem, and we can reduce the memory space by about half. When solving an SDP problem in which ELEMENTS is the most time-consuming part, we recommend the use of SDPARA 7.4.0 since the disadvantage (the increase in the total computation time) outweighs the advantage (the reduction of the total computational memory).

VI. TSUBAME 2.0: GPU-ACCELERATED SUPERCOMPUTER

This section briefly describes the TSUBAME 2.0 supercomputer, which was used for our evaluation of SDPARA and the new techniques introduced in this paper. It is to be noted that SDPARA and the new techniques were developed for general GPU clusters.

TSUBAME 2.0 is a GPU-based heterogeneous supercomputer with 2.4 PFlops of peak performance, and it is installed at the Tokyo Institute of Technology (for detailed information, refer to the technical paper [28]). The main part of the system consists of 1408 HP Proliant SL390s G7 nodes. With these nodes, the system has 16,896 cores, 4224 GPU devices, over 74 TB of memory capacity, and interconnection with 200 Tbps of bisection bandwidth.

Figure 7 shows the structure of each node, which has two Intel Xeon X5670 2.93 GHz (6 cores) CPUs, three NVIDIA Tesla M2050 GPUs, 54 GB (partly 96 GB) of DDR3 memory, and 120 GB SSDs as node local storage. A node also has QDR 40 Gbps InfiniBand HCAs connected to the dual-rail interconnection with full-bisection fat-tree topology. The two CPUs share 54 GB memory, and their total theoretical peak performance (double precision) is 140.8 GFlops. On the other hand, each M2050 GPU, whose peak performance (DP) is 515 GFlops, has discrete GDDR5 memory of 3 GB. CPUs and GPUs are connected via a PCI-Express 2.0 x16 bus. As described in Section VII, we carefully design the implementation by considering the characteristics of the CPUs and GPUs.

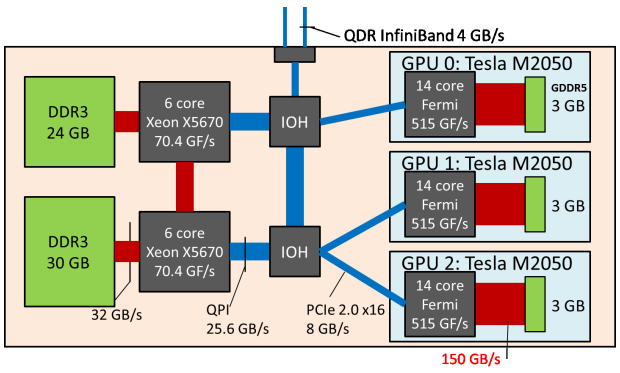


Fig. 7. Structure of the HP SL390s G7 node used in TSUBAME 2.0

VII. ACCELERATING CHOLESKY FACTORIZATION

A. Challenges

As described above, the total execution time of SDPARA is dominated by that of CHOLESKY (Cholesky factorization of SCM $\boldsymbol{B}$) when $\boldsymbol{B}$ is sufficiently large and dense. Generally, dense matrix computation can be significantly accelerated by harnessing GPGPU computing; for example, Endo et al. have shown that the performance of the Linpack benchmark (LU factorization with pivoting) can be scalably increased by using more than 1200 accelerators [29]. The keys to scalability include

- Overlapping computation on GPUs and PCI-Express (PCIe) communication between GPUs and CPUs.
- Configuring the *block size* nb so that data reuse is promoted and the amount of PCIe communication is reduced. In this paper, $nb = 1024$.

While the parallel Cholesky factorization algorithm and Linpack have many common points, the former poses new challenges because of the following differences.

- In each computation step, the part of the matrix to be updated is the lower-triangle part, rather than a rectangle. Upon two-dimensional block-cyclic distribution, the shape of the updated part in each process becomes more complex.
- (Related to the above difference) The computation amount per step is halved compared to LU factorization. This makes the computation / communication ratio even worse.
- Cholesky factorization requires additional work called "panel transposition."

B. Implementation and Optimization

Hereafter, we call SDPARA whose CHOLESKY component is accelerated by GPUs "SDPARA (version 7.5.0-G)." Our accelerated CHOLESKY component has properties similar to the `pdpotrf` function of ScaLAPACK. The dense matrix $\boldsymbol{B}$, with a size of $m \times m$, is distributed among MPI processes in the two-dimensional block-cyclic distribution with block size nb. When we let $mb = \lceil m/nb \rceil$ be the number of blocks that are aligned in a row or a column, the CHOLESKY algorithm consists of mb steps. A single (k-th) step proceeds as follows:

- **Diagonal block factorization:** The k-th diagonal block is Cholesky-factorized locally. Then, the result block is broadcast to processes in the k-th process column.
- **Panel factorization:** The k-th block columns are called "panel" $\boldsymbol{L}$, and the panel is factorized by using the `dtrsm` BLAS kernel.
- **Panel broadcast and transposition:** We need to broadcast $\boldsymbol{L}$ row-wise, obtain the transposition of L, and broadcast $\boldsymbol{L}^t$ column-wise.
- **Update:** This is the most computation-intensive part. Each process updates its own part of the rest matrix, taking the corresponding part of $\boldsymbol{L}$ and $\boldsymbol{L}^t$. Let $\boldsymbol{B}'$ be the rest matrix. Then $\boldsymbol{B}' = \boldsymbol{B}' - \boldsymbol{L} \times \boldsymbol{L}^t$ is computed. Thus, the `DGEMM` BLAS kernel dominates the execution time.

Note that updating the lower-triangular part is sufficient; thus, we can omit the computation of the unused upper part.

The basic approaches we applied to accelerate this algorithm are as follows.

1) We invoke one MPI process per GPU to drive it, and thus, three processes per node are invoked on TSUBAME 2.0 nodes.
2) It is important to use the fast `DTRSM` and `DGEMM` BLAS kernels on GPUs. We used highly tuned kernels developed by NVIDIA, which are faster than those in the official CUBLAS library. The `DGEMM` kernel achieves about 350 GFlops (on-board speed) on a M2050 GPU, while CUBLAS `DGEMM` achieves about 300 GFlops [5].
3) On GPU clusters, we have to decide where the data structure is located since the GPU device memory is separated from the host memory. In order to accommodate larger sizes of $\boldsymbol{B}$, we store it on the host memory.

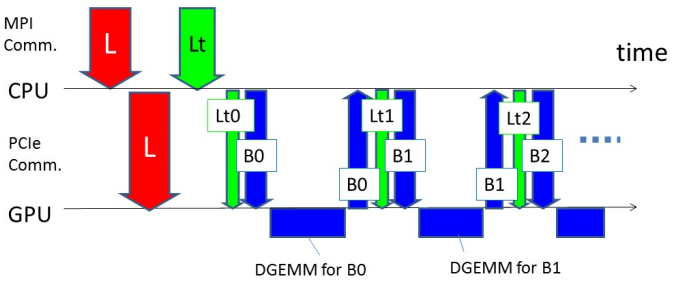
Approaches 2) and 3) indicate that we need to divide the matrices into parts smaller than the device memory capacity and send the input matrices to the GPU via PCIe in order to perform partial computation. Of course, without overlapping computation and communication (as in "version 1" in Figure 8), the performance is strictly restricted. To ensure high performance, the following optimization and configuration are adopted.

- When the size of the partial matrix to be updated by a single GPU is $r \times s$, the computation cost in the "update" phase is $O(r \cdot s \cdot nb)$, while the communication cost is $O(r \cdot s + r \cdot nb + s \cdot nb)$. To reduce the relative communication cost, the block size nb should be sufficiently large. Here there is a trade-off since very large nb degrades load balancing. After a preliminary evaluation, we set nb to be 1024.
- In order to reduce and hide the PCIe communication cost, we overlap GPU computation and PCIe communication.

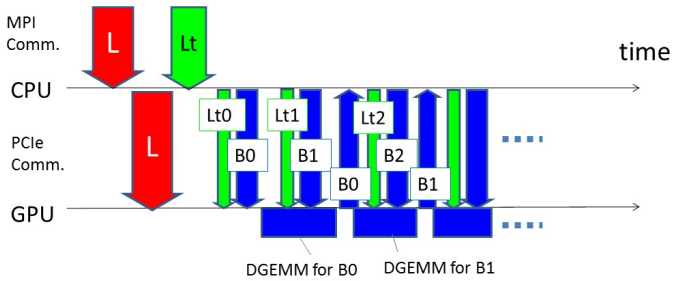
With these methods ("version 2" in the figure) we can enjoy the accelerated performance. However, we have noticed that inter-node MPI communication costs still restrict the scalability. Although each node in TSUBAME 2.0 has a wide injection bandwidth (8 GB/s), the communication costs increase relatively when the computation is accelerated. Here, we further promote the overlapping policy described above; we overlap all computations, PCIe communication, and MPI communication ("version 3" in the figure). For this, the "Panel broadcast and transposition" and "Update" phases are reorganized; the transposed panel $\boldsymbol{L}^t$ is divided into pieces before broadcasting, and each process transfers them to the GPU just after partial broadcast is completed.

As future improvement, we could overlap MPI communication for $\boldsymbol{L}$ and computation by using an optimization technique called "lookahead," which has been introduced in High-Performance Linpack [30].

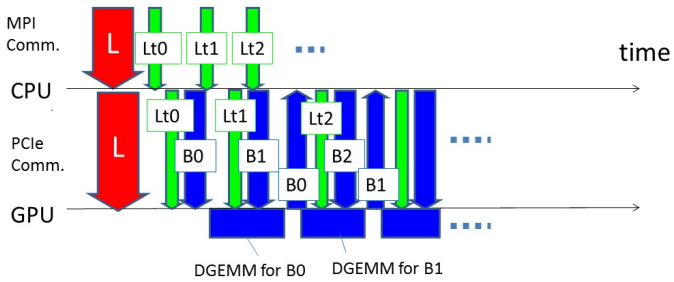
[5]Unfortunately, the speed of 350 GFlops is still far from the theoretical peak of 515 GFlops. NVIDIA staff mention that this is because of the limitation of the architecture of Fermi generation GPUs and that the speed will increase in next-generation GPUs.



Version 1: No overlapping



Version 2: GPU computation and PCIe communication are overlapped



Version 3: GPU computation, PCIe communication, and MPI communication are overlapped

Fig. 8. Several versions of the Cholesky factorization algorithm

VIII. Performance Evaluation

We have conducted the performance evaluation of SDPARA 7.5.0-G with GPU-accelerated Cholesky factorization on the TSUBAME 2.0 supercomputer. Table V shows the problems considered in the evaluation; the problems have characteristics that lead to the SCM being dense. We used NVIDIA CUDA 4.0, MVAPICH2 1.5.1, and Intel compiler 11.1 as the underlying system software. We used BLAS kernels from NVIDIA

TABLE V
PROBLEMS CONSIDERED IN THE EVALUATION

Name	SCM size (= m)	Application area
QAP4	211,926	QAP
QAP5	493,056	QAP
QAP6	709,275	QAP
QAP7	1,218,400	QAP
Truss	1,141,303	Truss topology design
QAP8	1,484,406	QAP

for use on GPUs, as mentioned in SectionVII-B.

We use different MPI processes for the different GPUs; thus, three processes run on a TSUBAME 2.0 node. A single process is configured to drive a GPU and three CPU cores, by considering the CPU affinity by using `cudaSetDevice` CUDA API and `sched_setaffinity` systemcall; for example, a process that uses GPU 0 in each node is bound to the cores in CPU 0 (Figure 7). Three CPU cores are used in INIT, ELEMENTS, and so on, while they are unused in the `DGEMM` or `DTRSM` kernels in CHOLESKY. For these kernels, only GPUs are used [6].

A. Performance of Accelerated Cholesky Factorization

First, the measured performance for a moderate number of nodes is shown in Figure 9. The graph shows the speed of the CHOLESKY component in TeraFlops. We see that its speed for the *QAP5* problem reaches 46.5 TFlops for 64 nodes (192 GPUs). This high performance is obviously impossible without GPUs since the theoretical peak performance of CPUs on 64 nodes is $64 \times 0.1408 = 9.01$ TFlops. Compared to the peak performance of CPUs and GPUs ($64 \times 1.686 = 107.9$ Tflops), the relative performance is 43.1%. This gap results from several factors, and the most important one is the limited performance of on-board DGEMM (only 68% of peak), which will be improved in next-generation Kepler GPUs. A comparison between *QAP5* and *QAP4* shows that the former shows better Flop performance since a larger matrix size is advantageous in CHOLESKY with $O(m^3)$ computation and $O(m^2)$ communication.

The graph also shows the effect of overlapping MPI communication by presenting a comparison between two CHOLESKY implementations: Version 3 and Version 2 described in Section VII-B. By comparing *QAP4/v3* and *QAP4/v2*, we can see that the effect of overlapping increases with the larger number of nodes; the difference is 10% for 64 nodes.

Figure 10 shows the performance of CHOLESKY on almost all nodes of TSUBAME 2.0 (1,360 nodes with 4,080 GPUs). For the *QAP8* problem, where the SCM size m is $1,484,406$, we achieve 533TFlops, which is the maximum speed achieved to date for a mathematical optimization problem. As in the previous graph, we see that a larger problem corresponds to a higher speed.

[6]We made this decision on the basis of the preliminary evaluation, which showed that using both GPUs and CPUs slightly degraded the total performance. We consider this to be a result of the increased use of the CPU bus and memory bandwidth, which will be investigated in detail later.

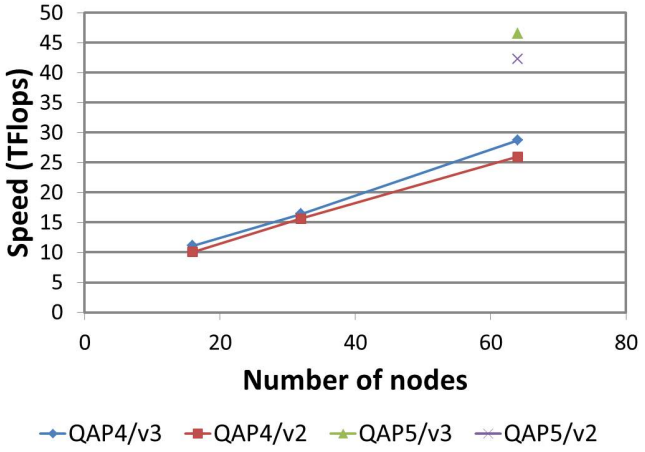


Fig. 9. Performance of GPU Cholesky factorization obtained by using up to 64 nodes (192 GPUs). Version 3 and Version 2 of CHOLESKY are compared

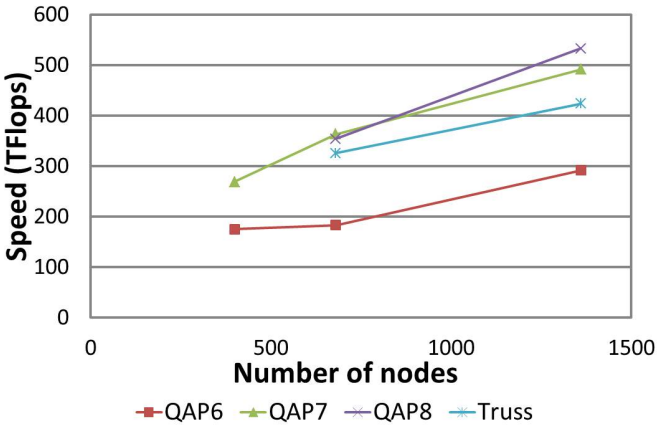


Fig. 10. Performance of GPU Cholesky factorization on the full TSUBAME 2.0 system (up to 1,360 nodes and 4,080 GPUs)

Since the peak performance of 1,360 nodes is 2,293 TFlops, the relative performance with *QAP8* is 24%. This is lower than results shown in the previous graph. However, we note that there is plenty of space in the node memory owing to the optimization for memory usage reduction. In the case of *QAP8* on 1,360 nodes, only 21 GB of the 54 GB memory is used per node. Thus, we can accommodate larger a problem with $m \sim 2,000,000$, which would further increase the speed.

B. Performance of SDPARA Solver

In addition to CHOLESKY, we have to eliminate any bottlenecks hidden in the entire SDPARA solver to solve real problems in a scalable manner. During the preliminary evaluation, the INIT component (the initialization of SDPARA) becomes a bottleneck when the problem size and the number of nodes used are large. While the ELEMENTS components and other components have been parallelized in the original

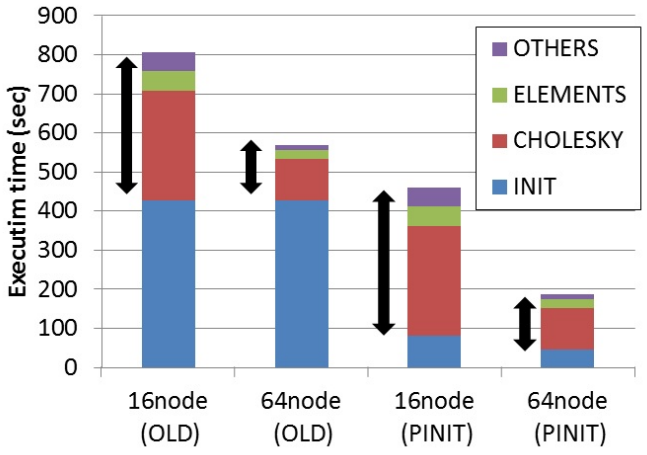


Fig. 11. Time breakdown of the SDPARA solver when QAP4 is solved. For CHOLESKY, ELEMENTS and OTHERS, execution times for a single iteration are shown. "PINIT" means "parallel initialization."

version, INIT has been done inefficiently, with all the MPI processes doing the same work corresponding to $O(m^2)$. Figure 11 shows the breakdown of the execution time for the different components. When we compare the case of *16 nodes (OLD)* and *64 nodes (OLD)*, we see that the relative cost for INIT is higher in the later.

The reasons we have overlooked this cost are as follows. (1) Unlike other components, INIT, which is out of the convergence loop, is executed only once. (2) Until CHOLESKY has been accelerated by the GPUs, INIT was not seen as bottleneck. However, with accelerated CHOLESKY, problems with $m > 10^6$ are now realistic; for example, when $m = 1,484,406$ (*QAP8*), inefficient INIT would take about 6 h. By parallelizing the phase, the situation is improved, as observed in the case of *16/64nodes (PINIT)* in the graph.

Table VI shows the time breakdown when *QAP8* is solved with 1,360 nodes. We see that during a single iteration of 2,712 s, 75% of the time is consumed by CHOLESKY. With the improved version of INIT, the INIT phase takes only ten minutes. If we use the original version, the 6-h INIT would make the evaluation impractical. Another observation is that the next target for acceleration would be ELEMENTS, which accounts for 14% of the iteration time.

C. Discussion

As described in Section VII, our CHOLESKY with GPUs adopts an approach similar to ScaLAPACK, with a drastic extension required to harness the high performance provided by thousands of GPUs. Recently, other approaches based on DAG schedulers of fine-grained tasks have been proposed,

TABLE VI
TIME BREAKDOWN OF SDPARA ON THE FULL TSUBAME 2.0 FOR QAP8.

Component	Time (s)
INIT	573
CHOLESKY	2,045
ELEMENTS	388
OTHERS	279

such as PLASMA[31], DAGuE[32], and StarPU[33]. While most of them are for single-node environments, DAGuE is designed for multiple nodes, and Bosilca has evaluated the performance of Cholesky factorization with GPUs on it [34]. However, the number of nodes used is limited to less than 10, and the scalability on thousands of GPUs is not shown. Further, there seems to be another issue pertaining to task granularity. Since their approach divides the computation tasks into tile levels (similar to blocks), each task is so small that it is difficult to harness the performance of BLAS kernels sufficiently especially on GPUs.

Despite these issues, it would be interesting to combine our implementation and the DAG-based approach is a future study since DAG schedulers are expected to enable communication/ computation overlap without complex hand coding, similar to the case in the present study.

IX. CONCLUSION

This paper described a high-performance solver, SDPARA 7.5.0-G, for large-scale, SDP problems. The key for high performance is the acceleration of Cholesky factorization by using thousands of GPUs. With 4,080 NVIDIA M2050 GPUs on the TSUBAME 2.0 supercomputer, our implementation achieved 533 TFlops in double precision for a large-scale problem with $m = 1,484,406$. Since SDPARA is a general solver for real problems, we not only improved the dense matrix computation component but also reviewed the entire software package to eliminate bottleneck parts that have previously been hidden. In the review, we also made the following improvements: parallelization of the solver initialization phase (INIT) and modification of the data structure used for the generation of Schur complement matrix (ELEMENTS). With all these improvements, we have demonstrated that solving SDP problems with $m > 10^6$ is now possible on modern accelerated supercomputers.

ACKNOWLEDGMENT

This research was supported by the Japan Science and Technology Agency (JST), the Core Research of Evolutionary Science and Technology(CREST) research project, and the TSUBAME 2.0 Supercomputer Grand Challenge Program at the Tokyo Institute of Technology. The authors would like to thank Prof. Hayato Waki for providing us with SDP problems for QAP.

REFERENCES

[1] Handbook of Semidefinite Programming (International Series in Operations Research and Management Science), edited by H. Wolkowicz, R. Saigal, L. Vandenberghe, Kluwer Academic Pub, (2000)

[2] Handbook on Semidefinite, Conic and Polynomial Optimization (International Series in Operations Research and Management Science), edited by M.F. Anjos and J.B. Lasserre, Springer Science+Business Media, (2011)

[3] M. Yamashita, K. Fujisawa, M. Fukuda, K. Kobayashi, K. Nakata, M. Nakata: Latest Developments in the SDPA Family for Solving Large-Scale SDPs, M.F. Anjos and J.B. Lasserre (eds.), Handbook on Semidefinite, Cone and Polynomial Optimization: Theory, Algorithms, Software and Applications, Springer Press, Chapter 24, 687–713, (2011)

[4] K. Fujisawa, M. Kojima, K. Nakata: Exploiting sparsity in primal-dual interior-point methods for semidefinite programming. Math. Prog., **79**, 235–253 (1997)

[5] K. Fujisawa, K. Nakata, M. Yamashita, M. Fukuda: SDPA Project: Solving large-scale semidefinite programs. J. Oper. Res. Soc. Japan, **50**, 278–298 (2007)

[6] B. Borchers: CSDP, a C library for semidefinite programming. Optim. Meth. Softw., **11 & 12**, 613–623 (1999)

[7] J.F. Sturm: SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optim. Meth. Softw., **11 & 12**, 625–653 (1999)

[8] K.-C. Toh, M.J. Todd, R.H. Tütüncü: SDPT3 ? MATLAB software package for semidefinite programming, version 1.3. Optim. Meth. Softw., **11 & 12**, 545–581 (1999)

[9] M. Yamashita, K. Fujisawa, M. Kojima: SDPARA: SemiDefinite Programming Algorithm paRAllel version. Parallel Comput., **29**, 1053–1067 (2003)

[10] M. Yamashita, K. Fujisawa, M. Fukuda, K. Nakata and M. Nakata, Parallel solver for semidefinite programming problem having sparse Schur complement matrix, To appear in ACM Transactions on Mathematical Software (2012)

[11] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1997)

[12] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods in Appl. Mech. Eng., **184**, 501–520 (2000)

[13] I. D. Ivanov, E. de Klerk: Parallel implementation of a semidefinite programming solver based on CSDP in a distributed memory cluster. Optim. Meth. Softw., **25(3)**, 405–420 (2010)

[14] S. J. Benson: Parallel computing on semidefinite programs *Preprint ANL/MCS-P939-0302* (2002)

[15] M. Kojima, S. Shindoh, S. Hara: Interior-point methods for the monotone semidefinite linear complementarity problems. SIAM J. Optim., **7**, 86–125 (1997)

[16] M. Ohsaki, K. Fujisawa, N. Katoh, Y. Kanno: Semi-definite programming for topology optimization of truss under multiple eigenvalue constraints. The Computer Methods in Applied Mechanics and Engineering, **180**, 203–217 (1999).

[17] S. Burer, "On the copositive representation of binary and continuous nonconvex quadratic programs", *Mathematical Programming*, 120, 479–495 (2009)

[18] Q. Zhao, S. E. Karisch, F. Rendl and H. Wolkowicz, "Semidefinite Programming Relaxations for the Quadratic Assignment Problem", *Journal of Combinatorial Optimization*, 2 71–109, 1998.

[19] P.A.M. Dirac: The Quantum Theory of the Electron. Roy. Soc. (London), **A 123**, 714 (1929)

[20] A.J. Coleman: Structure of fermion density matrices. Rev. Mod. Phys. **35**, 668–687 (1963)

[21] C. Garrod, J.K. Percus: Reduction of the N-particle variational problem. J. Math. Phys. **5**, 1756–1776 (1964)

[22] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, K. Fujisawa: Variational calculations of fermion second-order reduced density matrices by semidefinite programming algorithm. J. Chem. Phys. **114**, 8282–8292 (2001)

[23] Z. Zhao, B.J. Braams, M. Fukuda, M.L. Overton, J.K. Percus: The reduced density matrix method for electronic structure calculations and the role of three-index representability conditions. J. Chem. Phys. **120**, 2095–2104 (2004)

[24] F. Alizadeh, J.-P.A. Haeberly, M.L. Overton: Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results. SIAM J. on Optim., **8**, 746–768 (1998)

[25] C. Helmberg, F. Rendl, R.J. Vanderbei, H. Wolkowicz: An interior-point method for semidefinite programming. SIAM J. Optim., **6**, 342–361 (1996)

[26] R.D.C Monteiro: Primal-dual path following algorithms for semidefinite programming. SIAM J. Optim., **7**, 663–678 (1997)

[27] Y.E. Nesterov, M.J. Todd: Primal-dual interior-point methods for self-scaled cones. SIAM J. Optim, **8**, 324–364 (1998)

[28] S. Matsuoka, T. Endo, N. Maruyama, H. Sato, S. Takizawa: *The Total Picture of TSUBAME2.0*. TSUBAME e-Science Journal, GSIC, Tokyo Institute of Technology, Vol. 1, 2–4 (2010)

[29] T. Endo, A. Nukada, S. Matsuoka, N. Maruyama: Linpack Evaluation on a Supercomputer with Heterogeneous Accelerators. Proceedings of IEEE IPDPS10, 1–8 (2010)

[30] A. Petitet, R.C. Whaley, J. Dongarra, A. Cleary: HPL ? A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, `http://www.netlib.org/benchmark/hpl/`

[31] J. Kurzak, H. Ltaief, J. Dongarra, R. Badia: Scheduling dense linear algebra operations on multicore processors. Concurrency and Computation: Practice and Experience, **22**, no. 1, 15–44 (2010)

[32] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra: DAGuE: A generic distributed DAG engine for high performance computing, Innovative Computing Laboratory Technical Report, ICL-UT-10-01 (2010)

[33] C. Augonnet, S. Thibault, R. Namyst, P. A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures: Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science, **5704**, 863–874 (2009)

[34] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarier, N. Saengpatsa, S. Tomov, J. Dongarra: Performance portability of a GPU enabled factorization with the DAGuE framework. IEEE Cluster: workshop on Parallel Programming on Accelerator Clusters (PPAC) (2011)